Ordenação Paralela com OpenMP e CUDA

Heitor Colichio, Yago Pimenta, Pedro Borges, Matheus Menecucci, Luiz Barros, Hélio Guardia

Departamento de Computação - Universidade Federal de São Carlos

{heitorcolichio, matheus.menecucci, yagopimenta, luizbarros}@estudante.ufscar.br

Resumo. Este artigo apresenta um estudo da paralelização do problema da ordenação de valores. Para tanto, parte-se de uma versão do algoritmo QuickSort e explora-se o particionamento dos dados para ordenação de listas parciais, com a consequente integração dos resultados, de maneira semelhante às estratégias do algoritmo MergeSort. Duas implementações são apresentadas, uma para ambiente multiprocessado com memória compartilhada, programado com OpenMP, e outra em GPU, utilizando CUDA. Os resultados obtidos mostram a viabilidade do uso do paralelismo neste problema, com *speedups* significativos.

1. Introdução

QuickSort é um algoritmo de ordenação sequencial amplamente utilizado, reconhecido por sua eficiência média O (nLogN) [Thomas et al., 2009]. Embora este algoritmo seja eficaz sequencialmente, este trabalho visa a investigar aspectos de sua paralelização no tratamento de grandes conjuntos de dados não ordenados. Para tanto, duas abordagens são utilizadas: paralelismo de *threads* com OpenMP, em ambiente multicore, e CUDA, para programação em GPU. Considerando a estratégia de divisão e conquista tipicamente utilizada em algoritmos de ordenação, OpenMP oferece uma maneira eficiente de criar tarefas de forma dinâmica e de sincronizar a execução dessas tarefas [OpenMP, 2021]. CUDA, por sua vez, proporciona uma execução paralela em granularidade fina, com potencial para acelerar algoritmos intensivos em computação [Nvidia, 2023]. O tratamento de operações iterativas em GPU, contudo, requer cuidado para evitar que comunicações entre o espaço de endereçamento em memória RAM e as áreas de memória da GPU não se tornem um gargalo para o desempenho do programa paralelo.

Os resultados obtidos indicaram significativa melhoria do tempo gasto e do *speedup*, no ambiente multicore. Com CUDA, pequenas melhorias também foram notadas, ao custo de alterações na forma de tratamento dos resultados parciais gerados pelos blocos paralelos.

2. Ordenação com QuickSort

Entre os algoritmos de ordenação mais conhecidos, *QuickSort*, apresentado na Listagem 1, destaca-se por sua simplicidade e eficiência, sendo baseado em comparações e utilzando uma abordagem de divisão e conquista.

Listagem 1: Exemplo de implementação - *QuickSort*

```
void quick_sort(int *a, int left, int right) {
  int i, j, x, y; i = left; j = right; x = a[(left + right) / 2];
  while (i <= j) {
    while (a[i] < x && i < right) i++;
    while (a[j] > x && j > left)  j--;
    if (i <= j) { y = a[i]; a[i] = a[j];  a[j] = y; i++;  j--;
  }
  if (j > left) quick_sort(a, left, j);
  if (i < right) quick_sort(a, i, right); }
```

A eficiência do QuickSort reside principalmente em sua complexidade média de tempo de O (n log n), e seu desempenho superior é atribuído à estratégia de divisão e conquista [Hoare et. al., 1962]. Seu pior caso é O (n^2), que ocorre quando a escolha do pivô não é ideal e resulta em divisões desequilibradas dos subconjuntos.

A necessidade de tratar grandes volumes de dados motiva a exploração de paralelismo no processamento. A estratégia principal é dividir o trabalho em tarefas menores que são distribuídas entre as *threads* disponíveis, permitindo que várias partes do vetor sejam ordenadas simultaneamente. Outras abordagens para o mesmo problema podem serem vistas em [Satish et. al. 2009], onde os autores exploram técnicas como a divisão do trabalho em blocos menores para aproveitar o paralelismo massivo oferecido pelas GPUs, juntamente com otimizações de memória e acesso aos dados para minimizar gargalos de comunicação e maximizar a utilização dos recursos da GPU.

3. Ordenação Paralela

A estratégia de paralelização do código OpenMP é apresentada na Listagem 2.

Listagem 2 - Trecho do código paralelizado com OpenMP

```
void Qsort(char **arr, int left, int right) {
   if (left < right) {
       int mid = left + (right - left) / 2;
       if (right - left + 1 <= TASK_SIZE)
          Qsort(arr, left, right, mid);
       else {
           #pragma omp task
           Qsort(arr, left, mid);
           #pragma omp task
           Qsort(arr, mid + 1, right);
           #pragma omp taskwait
           merge(arr, left, mid, right);
}  }  }

... //  Na função main: criação da região paralela e do time de threads.
#pragma omp parallel
    #pragma omp single // Apenas 1 thread do time invoca a função inicial
    Qsort(strings, 0, num_elem - 1);
```

Na Listagem 2, as chamadas do ***Qsort*** são paralelizadas usando a diretiva ***#pragma omp task***, indicando ao compilador que cada ordenação parcial deverá ser tratada por uma tarefa independente. Estas são adicionadas a uma fila de execução e sincronizadas antes da mesclagem das partes ordenadas do vetor. Ao utilizar a diretiva ***#pragma omp task***, as chamadas recursivas da função Qsort são transformadas em tarefas que podem ser executadas em paralelo por qualquer uma das *threads* do time criado com a região paralela. Na função *main() ocorre a criação da região paralela e do seu respectivo time de threads para executar a função de ordenação (Qsort())*. Como a abordagem utilizada neste estudo foi baseada na criação de tarefas dinâmicas (*tasks*), apenas uma das *threads* da região paralela irá realizar a chamada inicial, o que é garantido com a diretiva ***single***.

Na solução desenvolvida, o grau de paralelismo da região paralela foi definido via variável de ambiente OMP_NUM_THREADS. Além disso, para que o número de tarefas (*tasks*) a serem executadas por essas *threads* não se tornasse um gargalo, a variável TASK_SIZE foi utilizada para determinar o momento de parada das invocações recursivas e da criação de novas *tasks*.

As estratégias para paralelização com ***CUDA*** são apresentadas na Listagem 3 e no recorte da função *main()*, onde ocorrem as invocações do *kernel*. A solução adotada divide o conjunto de dados em segmentos menores e mescla esses segmentos em ordem

crescente por meio de um *kernel* CUDA. O processo de ordenação combina segmentos de tamanho crescente dos dados. Os resultados são transferidos de volta para a CPU apenas ao final da ordenação, evitando cópias de memória nas etapas intermediárias. Para tanto, utiliza-se 2 vetores mantidos na área de memória da GPU, alterando-se os dados entre eles a cada etapa do processo.

Listagem 3: função merge Kernel

```
__global__ void mergeKernel(char* arr, char* aux, int tamAtual, int n_items, int width)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x; int low = idx * width;
        if(low >= n_items - tamAtual || low < 0) return; // evitar índices inválidos
    int mid = low + tamAtual - 1;
    int high = min(low + width - 1, n_items - 1); // evitar  high maior que o vetor
    merge (arr, aux, low, mid, high, MAX_STRING_SIZE); // função de merge na GPU
}
int main() {
 // Leitura dos dados (strings)
 cudaMalloc((void **)&d_data, n_items * MAX_STRING_SIZE);
 cudaMalloc((void **)&auxArr, n_items * MAX_STRING_SIZE);
 cudaMemcpy(d_data, h_data, n_items * MAX_STRING_SIZE, cudaMemcpyHostToDevice);
 for(int tamAtual = 1; tamAtual < n_items; tamAtual *= 2) { // invocações do kernel
  int width = tamAtual*2; int numSorts = (n_items + width - 1)/width;
  int threadsPerBlock = 64;
  if(numSorts < 32) { threadsPerBlock = 2; }
  int blocksPerGrid = (numSorts + threadsPerBlock - 1) / threadsPerBlock;
  mergeKernel<<<blocksPerGrid,threadsPerBlock>>>(d_data,auxArr,tamAtual,n_items, width);
  char* tmp = d_data; //troca de ponteiros para manter os dados na GPU
  d_data = auxArr; auxArr = tmp;
}
```

Na função *main()*, após a leitura do arquivo com as *strings* que serão ordenadas, é alocado espaço na GPU para o vetor auxiliar e para o vetor com as *strings* que serão ordenadas. Como os dados estão no espaço de memória da CPU, eles são copiados para a GPU com ***cudaMemcpy***, utilizando a flag ***cudaMemcpyHostToDevice***. Em seguida, é feita a definição do *grid* de *threads*. Nas *threads* por bloco, foi escolhido 64, em função do *warp size* (32), reduzindo para 2 quando houver poucas operações de merge. Depois, calcula-se a quantidade de blocos por *grid* fazendo com que cada *thread* fique responsável por uma operação de *merge*. Por fim, o *kernel* é chamado, passando os argumentos da função e o tamanho do grid. Um aspecto importante da solução CUDA desenvolvida é a estratégia de manter na memória da GPU o vetor sendo ordenado. Para isso, usa-se um vetor auxiliar na GPU, que recebe os resultados intermediários a cada rodada de ordenação parcial e junção dos resultados.

4. Resultados

As avaliações das estratégias de paralelização foram realizadas em 2 sistemas computacionais distintos, um computador com processador Intel Xeon Silver 4208 com 16 cores físicos e 32 cores lógicos, para testar a versão OpenMP, e *Google Collab*, usando uma GPU *Tesla* T4, para os testes com CUDA.

Os resultados de execuções da versão da ordenação paralelizada com OpenMP para 10.000.000 strings e TASK_SIZE=100 são apresentados nas Figuras 2 e 3.

Em *CUDA*, o uso de 32 *threads* por bloco gerou melhores resultados, embora o número de *threads* ativas seja reduzido para apenas 2 quando há poucas operações de *merge* (Tabela 1). A quantidade de blocos por grid é calculada de forma que cada *thread* fique responsável por uma operação de *merge*. Os testes permitiram explorar melhorias significativas no tempo de execução para a ordenação de grandes conjuntos de dados, com um speedup de 1,74 no pior caso (256 threads) e 2,62 no melhor caso (32 threads).

Figura 2 - *Tempo gasto*

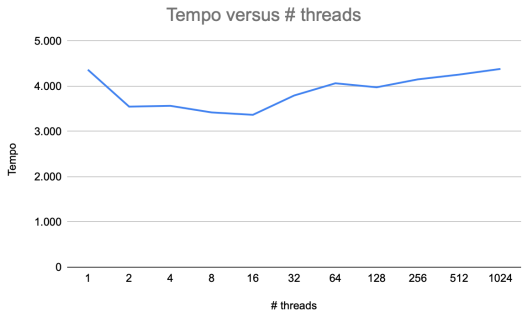

Figura 3 - *Speedup com OpenMP*

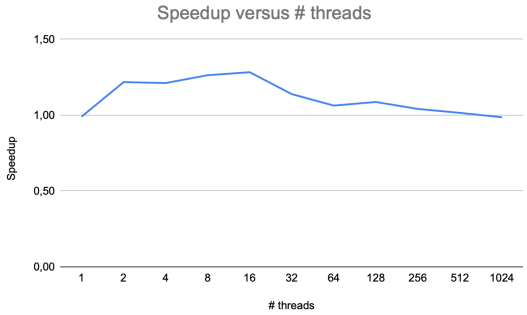

Tabela 1: Resultados da execução em GPU

Threads per block	Tempo execução	Speedup
16	1.934s	2,23
32	**1.648s**	2,62
64	1.654s	2,61
128	1.726s	2,50
256	2.483s	1,74

5. Conclusões

No ambiente multicore (OpenMP), foi possível observar uma melhora no *speedup* com a paralelização usando a criação de tarefas dinâmicas (*tasks*). A limitação das chamadas recursivas e da criação de tarefas (*tasks*) é importante para evitar sobrecargas com o paralelismo à medida que os dados particionados são muito reduzidos. Em CUDA, a manutenção dos dados dos vetores de elementos na memória da GPU durante as seguidas invocações do *kernel* colaborou para os ganhos de desempenho.

Como trabalhos futuros, pretende-se explorar variações no limite do grau de paralelismo, definido por TASK_SIZE no cenário com OpenMP, e também o grau de paralelismo dentro de cada bloco de *threads* CUDA, à medida que o número de operações de *merge* é reduzido.

6. Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). Cambridge, MA: MIT Press.

OpenMP (2021). Openmp application programming interface specification version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Acesso em 29 de março, 2024.

Nvidia (2023). Cudatoolkitdocumentation. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html. Acesso em 1 de Abril, 2024.

Hoare, C. A. R. (1962). Quicksort. Communications of the ACM, 4(7), 321-322.

Satish, N., Harris, M., & Garland, M. (2009). Designing efficient sorting algorithms for manycore GPUs. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-10).